Lab: Asynchronous Programming

Problems for exercises and homework for the "Java Advanced" course @ SoftUni.

Part I: Single and Multi-Threading

1. Single Thread

Create a **task** that prints the numbers from 1 to 10. **Start a thread** executing the task.

Optional: Add **`System.exit(1)`** at the end of your program.

Optional: Experiment with **`thread.join()`**

Examples

Input	Output
no input	1 2 3 4 5 6 7 8 9 10

Solution

Create a new **`Runnable`** that will define the code for the task:

```
Runnable task = () -> {
    for (int i = 1; i <= 10; i++) {
        System.out.print(i + " ");
    }
};
```

Create a new **`Thread`** that will execute the task

```
Thread thread = new Thread(task);
```

Start the thread:

```
thread.start();
```

Optional: Add **`System.exit(1)`** at the end of your program

```
System.exit(1);
```

Optional: Experiment with **`thread.join()`**

```
thread.join();
```

Example: If you include **`System.exit(1)`** and in the same time omit **`thread.join()`**, it is **possible** that the **main thread closes the additional thread** before the additional thread is done with its task

```
thread.start();
//thread.join();
System.exit(1);
```

```
"C:\Program Files\Java\jdk1.8.0_9
Process finished with exit code 1
```

Example: By including **`thread.join()`** it is guaranteed that the **main thread will wait for the thread it has started** (**`thread.join()`** blocks the calling thread)

```
thread.start();
thread.join();
System.exit(1);
```

```
"C:\Program Files\Java\jdk1.8.0_91
1 2 3 4 5 6 7 8 9 10
Process finished with exit code 1
```

2. Multi-Thread

Create a task that prints the numbers from 1 to 10. **Start 5 threads** executing the same task.

After each printing, add **`Thread.yield()`** statement. **Join all threads**.

Examples

Input	Output
no input	*(Output can vary)* [0] [0] [0] [0] [0] [1] [1] [1] [2] [3] [2] [1] …

Solution

Create a new **`Runnable`** which prints the numbers and yields after each print. **`Thread.yield()`** will make the effect of thread switching more obvious.

```
Runnable task = () -> {
    for (int i = 0; i < 10; i++) {
        System.out.printf("[%s] ", i);
        Thread.yield();
    }
};
```

Create an array for all 5 threads and for each of them start a new task:

```
Thread[] threads = new Thread[5];
for (int i = 0; i < 5; i++) {
    threads[i] = new Thread(task);
    threads[i].start();
}
```

Join all 5 threads:

```
for (Thread thread : threads) {
    thread.join();
}
```

3. Responsive UI

Create a program that prints the **primes from 0 to N**. Implement a **responsive UI**, e.g. user can stop the program at any time.

If stopped, show appropriate message

Examples

Input	Output
13	[2, 3, 5, 7, 11]... 5 primes calculated.
9999999 stop	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]... 169922 primes calculated.

Solution

Read N, the upper bound:

```
Scanner scanner = new Scanner(System.in);
System.out.print("n = ");
int to = Integer.valueOf(scanner.nextLine());
```

Create a method **printPrimes()** which you will use as a task:

```
static void printPrimes(int to) {
}
```

Create a **List<Integer>** for storing all prime numbers:

```
List<Integer> primes = new ArrayList<>();
for (int number = 0; number < to; number++) {
    if (isPrime(number)) {
        primes.add(number);
    }
}
```

Inside the **for** loop, define a condition for thread interruption:

```
for (int number = 0; number < to; number++) {
    if (isPrime(number)) {
        primes.add(number);
    }

    if (Thread.currentThread().isInterrupted()) {
        System.out.println("Interrupted...");
        break;
    }
}
```

Print some of the primes and the count of all primes you have discovered:

```
System.out.println(primes.stream()
        .limit(10)
        .collect(Collectors.toList()) + "...");
System.out.printf("%s primes calculated.", primes.size());
```

Implement the method **isPrime()** yourself. It should evaluate a single number:

```
static boolean isPrime(int number) {...}
```

In the **main()**, create a new task with **printPrimes()** and start it:

```
Runnable task = () -> printPrimes(to);
Thread thread = new Thread(task);
thread.start();
```

Create a loop for user input:

```
while (true) {
    String command = scanner.nextLine();

    if (command.equals("stop")) {
        thread.interrupt();
        break;
    } else {
        System.out.println("unknown command");
    }
}
```

- Wait for the thread to finish execution:

```
thread.join();
```

4. Benchmarking

Test every number in the range [0...N] if it is prime or not. Spread the calculation over 2 or 4 threads.

Benchmark and compare the difference over one thread. Benchmark both efficient and inefficient **isPrime()**.

Examples

Input	Output
1000	*(Output guaranteed to vary)* Execution time: 184503539
999999	*(Output guaranteed to vary)* Execution time: 3274639906

Solution

Read N, the upper bound

```
Scanner scanner = new Scanner(System.in);
System.out.print("n = ");
int to = Integer.valueOf(scanner.nextLine());
```

Create a **List<Integer>** with all numbers

```
List<Integer> numbers = new ArrayList<>();
for (int i = 0; i <= to; i++) {
    numbers.add(i);
}
```

Start a clock for benchmarking:

```
long start = System.nanoTime();
```

Create a new **ExecutorService** with a fixed thread pool

```
ExecutorService es = Executors.newFixedThreadPool(4);
```

Create a **Future[]** with the size of all numbers

```
Future[] results = new Future[numbers.size()];
```

Test each number

```
for (int i = 0; i < numbers.size(); i++) {
    Integer number = numbers.get(i);
    Future<Boolean> future = es.submit(() -> isPrime(number));
    results[i] = future;
}
```

Await all tasks to finish

```
es.awaitTermination(100L, TimeUnit.MILLISECONDS);
```

Stop the benchmark and **print the results**. Make sure to **shut down** the executor service

```
long total = System.nanoTime() - start;
System.out.println("Execution time: " + total);
es.shutdown();
```

If you want the result for each number, you can get it from the Future array

```
for (Future f : results) {
    System.out.println(f.get());
}
```

Part II: Resource Sharing

5. Transactions

Create a simple **BankAccount** class with the following characteristics:

- Properties:
  - **Integer balance**
- Methods:
  - **void deposit(int sum)**

Create a **multi-threaded program** that simulates 100 transactions, each deposing 100 times 1 to the balance.

Examples

Input	Output
no input	*(Output should vary)* 9559
no input	*(Output should vary)* 9905

Solution

Create the class

```
private static class Account {
    int balance;

    void add (int amount) {
        balance = balance + amount;
    }
}
```

Create constants for the **number of transactions** and for **number of operations per transaction**

```
final int transactions = 100;
final int operationsPerTransaction = 100;
```

Create an **instance of the class** and a **task**

```
Account account = new Account();
Runnable task = () -> {
    for (int i = 0; i < operationsPerTransaction; i++) {
        account.add(1);
        Thread.yield();
    }
};
```

Create a new thread for each transaction

```
Thread[] threads = new Thread[transactions];
for (int i = 0; i < transactions; i++) {
    threads[i] = new Thread(task);
    threads[i].start();
}
```

Join all threads

```
Thread[] threads = new Thread[transactions];
for (int i = 0; i < transactions; i++) {
    threads[i] = new Thread(task);
    threads[i].start();
}
```

Print the results

```
System.out.println(account.balance);
```

Start the program multiple times and observe the results

6. Thread Safe Transactions

Make the previous application **thread safe**, e.g. you should get the **same result every time**.

Examples

Input	Output
no input	*(Output should not vary)* 10000
no input	*(Output should not vary)* 10000

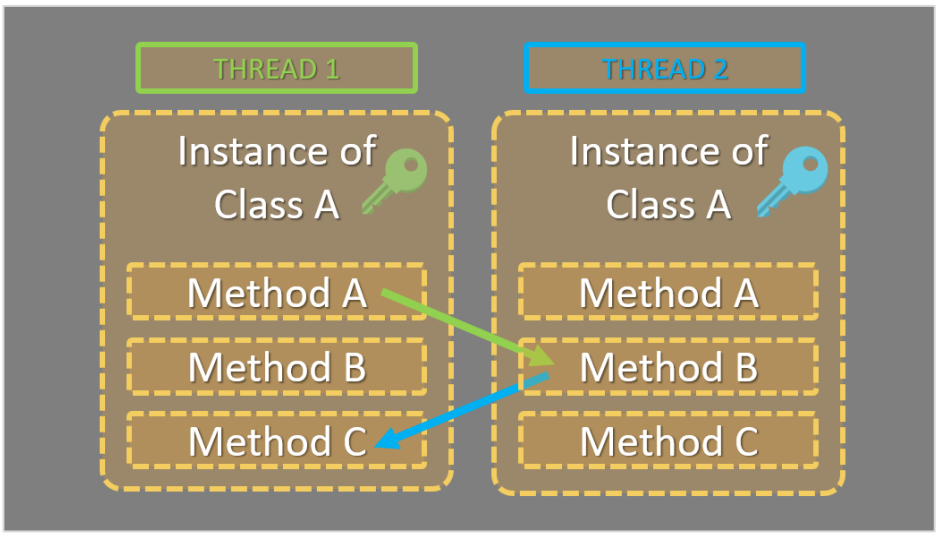
Solution

Make the method **add()** synchronized

```
synchronized void add (int amount) {
    balance = balance + amount;
}
```

7. * Deadlock

Reproduce the following **deadlock scenario**:



Solution

Create a class that will hold methods A, B and C

```
static class MyClass {
}
```

Add a property id and a constructor, setting the property

```
String id;

public MyClass(String id) {
    this.id = id;
}
```

Create a method **a()**, which should take a reference to the other instance of the class. Make sure it is declared with the **synchronized** keyword

```
synchronized void a(MyClass other) {
}
```

Print a message, that the method was called

```
System.out.printf("%s called method A on %s%n",
        this.id, other.id);
```

Sleep the thread for some milliseconds to ensure that the two methods will be called at the same time by the threads

```
try {
    Thread.sleep(100);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Call the **b()** method of the other instance and pass a reference to the current object

```
other.b(this);
```

Create the **b()** method, which should also print a message and call the other objects **c()** method

```
synchronized void b(MyClass other) {
    System.out.printf("%s called method A on %s%n",
            other.id, this.id);

    other.c();
}
```

Create the **c()** method

```
synchronized private void c() {
    System.out.println(this.id + " done");
}
```

In the main method, create two instances of the class

```
MyClass first = new MyClass("First");
MyClass second = new MyClass("Second");
```

Create two threads that start a new task

```
Thread tFirst = new Thread(() -> first.a(second));
Thread tSecond = new Thread(() -> second.a(first));
```

Start the threads

```
tFirst.start();
tSecond.start();
```

You should get a deadlock